



下载APP

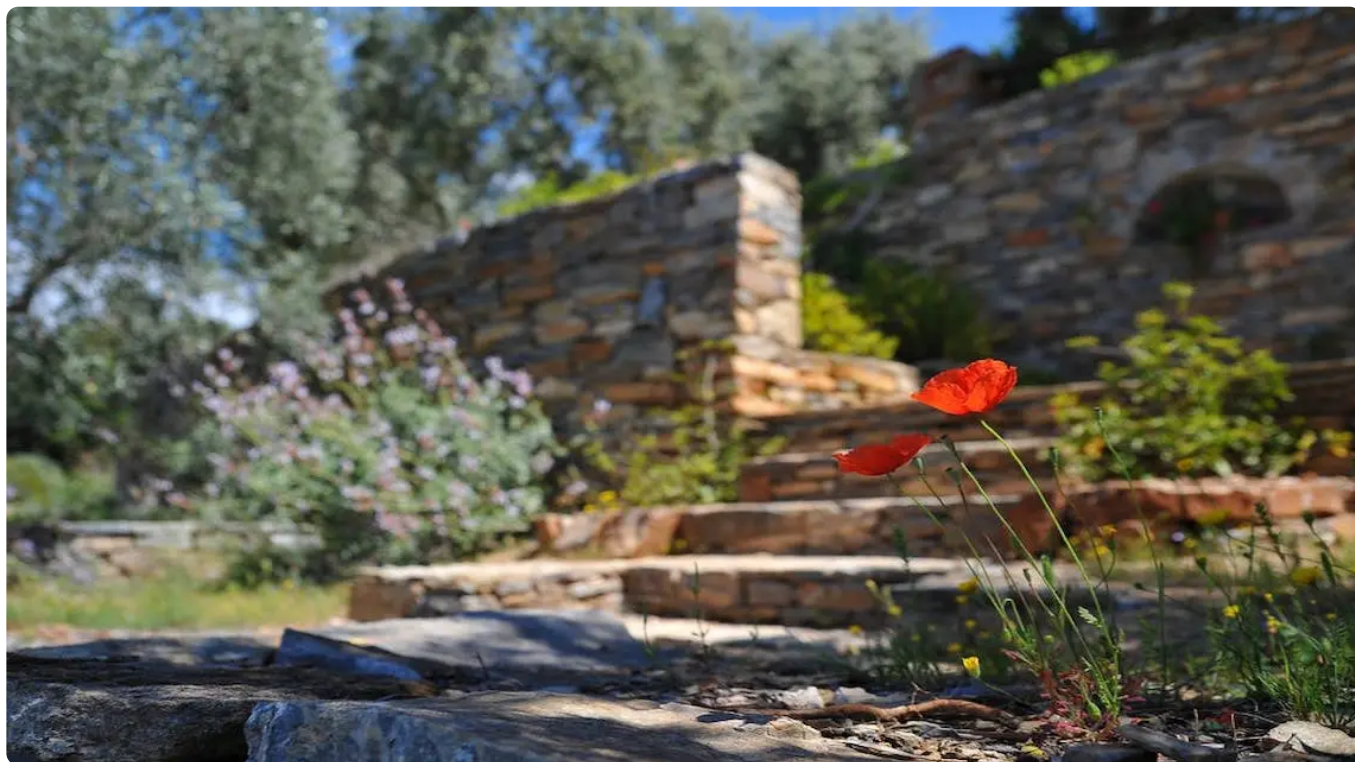


11 | 将模型实现为RESTful API (下)

2021-07-17 徐昊

《如何落地业务建模》

课程介绍 >



讲述：徐昊

时长 20:15 大小 18.55M



你好，我是徐昊。今天我们来聊聊继续如何将模型映射为 RESTful API。

上节课我们学习了如何将领域模型实现为 RESTful API 的宏观步骤，分为四步：

1. 通过 URI 表示领域模型；
2. 根据 URI 设计 API；
3. 使用分布式超媒体设计 API 中涉及的资源；
4. 使用得到的 API 去覆盖业务流程，验证 API 的有效性。



并着重学习了，如何通过 URI 表示领域模型；并在得到与领域模型对应的 URI 之后，使用类似角色 - 目标 - 实体法的分析方法，获得 API 候选。那么今天我们就继续学习后面两

步，通过分布式超媒体设计 API 中涉及的资源，并将使用 API 覆盖业务流程，以验证 API 的有效性。

RESTful API 是一种什么样风格的 API？

到目前为止我们得到的 API 候选，比如 GET /users 等，还不能被称作 RESTful API，而仅仅可以被叫做**基于 HTTP 的远程过程调用**（HTTP RPC）。那么 **RESTful API 到底是一种什么样的 API 呢？**

RESTful API 是指**符合 REST 架构风格的 API 设计**，而 REST 架构风格是对互联网规模架构（Internet Scale Architecture）的总结与提炼。这一切都源于 Roy Fielding 提出的一个问题：既然互联网（Internet）是人类迄今为止构造的最大的软件应用，那么到底是什么样的架构原则，支撑了如此规模的异构且互联的系统呢？我们能从中学习到什么，以帮助我们更好地构建软件？

Roy Fielding 将互联网定义为**分布式超媒体信息获取系统**（Distributed Hypermedia Information Retrieval System）。也就是由超媒体描述的、分布式的信息系统。在这个系统中，信息分布在不同的服务器中，并由超媒体联通。换句话说，**分布式超媒体是互联网的集成策略**。

因而我们必须理解这种集成策略给整个系统带来了什么影响，这样才能在实现 RESTful API 时，尽可能发挥它的优势。

在我看来，这种集成策略实现了客户端与服务器之间的渐进式服务消费（Progressive Service Consumption）。而这种渐进式服务消费，在客户端的多样性和 API 的稳定性之间，取得了完美的平衡。

也就是说，它让我们可以用同样一套 API，去对应具有不同诉求的参差多态的客户端们（我将这种 API 称为幸福 API，因为罗素说“参差多态乃是幸福的本源”。那么能够支持参差多态的客户端的 API，也就是幸福 API）。

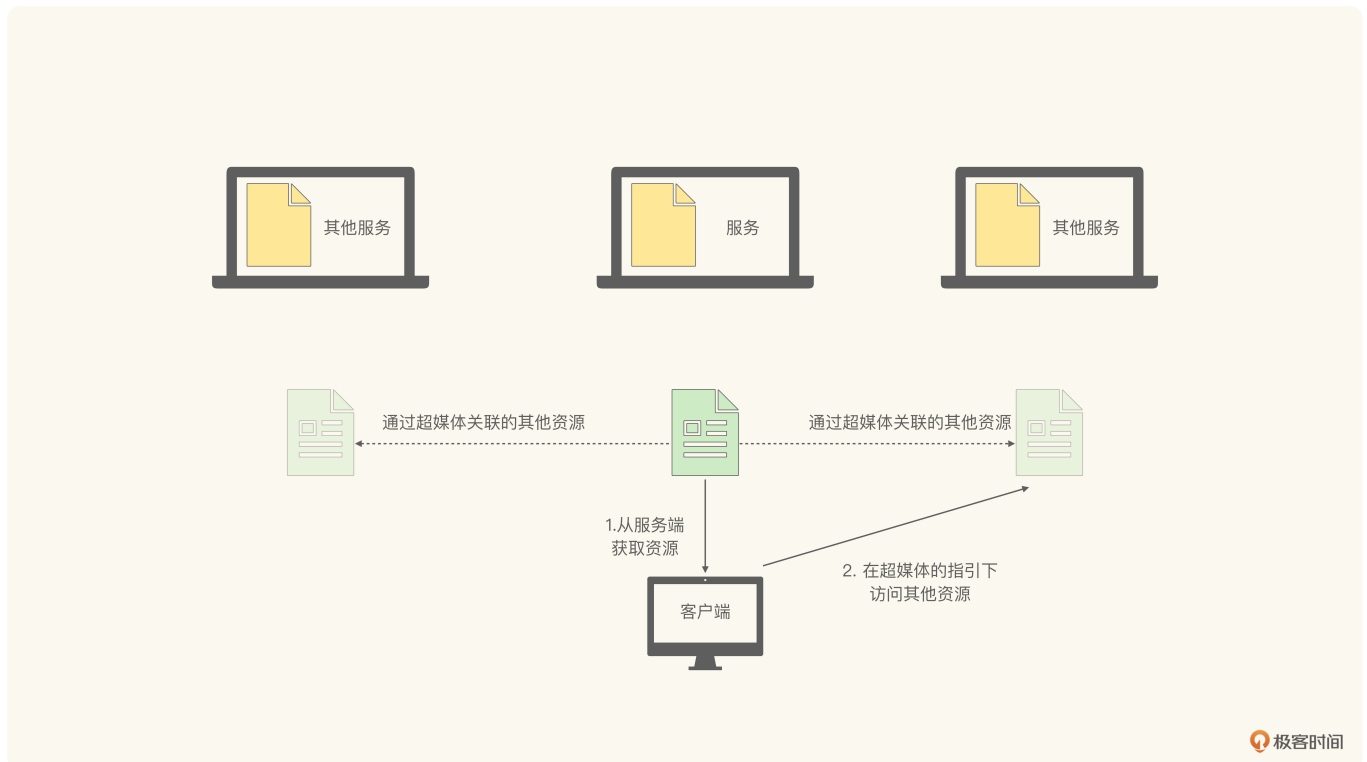
让我们回想一下，当你访问网络上的信息时，你会通过 URI 指定需要访问的页面。页面中会关联 CSS、JavaScript 等所需的资源，并包含其他页面的链接。浏览器会下载对应的 CSS、JavaScript，完成界面的渲染。你也可以通过页面中的链接，去访问其他关联的页面。

在整个过程中，你无需考虑 CSS、JavaScript 来自哪里，关联的页面是否来自同一个网站。因为从始至终，你都将这些关联的资源看作是一个完整的资源在消费。这是我们习以为常的场景。

然而如果我们从系统集成的角度重新看待这个过程，就会有不同的体会。**网页中对其他资源的引用，实际上表示了对其他服务计算结果的整合**。一个典型的例子就是 Google Analytics，虽然它是以 js 的形式出现。但当页面引用到了这个 js 资源时，实际上完成的是对 Google Analytics 服务的整合。

而当你点击页面中包含的链接时，浏览器会为你请求对应的网站资源，并将计算结果（大概率是另一个页面）在浏览器客户端里展示。这也可以看作是对其他系统的集成。更具体地讲，也就是浏览器客户端通过延迟（Lazy）调用的方式，整合了其他系统提供的服务。

所以无论是 css、js 这样的资源，还是通过超链接引用的其他页面，都是服务端返回的页面中描述的。而客户端则根据服务端提供的描述，选择恰当的时机，访问这些关联的资源，也就是整合其他系统提供的服务。这样的系统集成方式，我们称之为**由服务器指导的客户端侧集成**（Server-side guided client-side integration）。如图所示：



由于对于关联资源的访问是由客户端发起的，并不是由服务器端强制的，那么客户端可能出于某些原因（安全、计算资源受限等）选择不访问某些关联资源。

比如，在智能手机大规模流行之前，大部分手机上的浏览器不具备完整执行 js 的能力，因而很多浏览器选择不加载任何 js 资源。

从这样的角度出发，我们就会发现，CSS、JavaScript 或者页面中通过超链接引用的其他页面，**都可以看作是对当前页面的增强**。比如 CSS 增强了当前页面的视觉效果，js 增强了当前页面的交互，页面中的超链接为页面中的内容提供了额外的信息。**于是通过超媒体格式，我们不仅可以描述当前的服务，还可以描述增强的服务。**

比如，你在极客时间专栏里阅读这篇文章，那么当前的服务是让文章在浏览器中显示，供你阅读，这是默认服务。那么怎么增强它呢？我们可以提供语音的版本，也可以提供视频，还可以提供大字打印版等等。那么我们可以在超媒体格式中，描述这些关联的资源。如下 HTML 代码所示（HTML 是最常见的超媒体）：

[复制代码](#)

```
1 <article>
2   <title>11 | 将模型实现为RESTful API (下)</title>
3   <p>....</p>
4   <link href="http://voice.geekbang.org/chapter11.mp3" rel="voice"/>
5   <link href="http://video.geekbang.org/chapter11.mp4" rel="video"/>
6   <link href="http://easy-reading-print.com/13678328" rel="print"/>
7 </article>
```

在这段代码中，我们通过 link 标签，表示了这篇文章与其他资源的关系。可以看出，我们关联

了 “<http://voice.geekbang.org/chapter11.mp3>” “<http://video.geekbang.org/chapter11.mp4>” 和 “<http://easy-reading-print.com/13678328>”。它们与当前文章的关系分布为 voice、video 和 print。此时，客户端可以根据这些额外的信息，在不同的情景下选择加载哪些额外的资源。

假设此时我们三个客户端，分别是用于阅读的某阅读器，用于听书的某播放器，以及某个打印机。那么这些客户端只需要拿到这段超媒体，然后各自选择对应的服务即可：

阅读器按照 HTML 格式显示内容，忽略其他关联的服务；

播放器从超媒体中提取由 voice 链接的资源，去加载对应的 mp3，忽略其他 HTML 文本和服务；

打印机从超媒体中提取由print链接的资源，送去打印，忽略其他 HTML 文本和服务。

可以看到，我们使用了同样的 API，支撑起了完全不同的客户端。而这些客户端彼此之间也并不知道对方的存在，它们只关心自己关注的资源。换言之，**通过超媒体描述的增强服务，让客户端和服务端之间形成了一种协商与匹配的关系，也就是我们所说的渐进式服务消费。**

所谓渐进式，指得是按需索取，而不要求消费全部关联的服务。也正是渐进式的消费方式，促使互联网成为了一个异构、松耦合且开放的系统。同时这也正是我们希望自己所构建的系统能够具有的特质。

所以**通过分布式超媒体的设计，实现这种渐进式服务消费，也就是我们从 HTTP RPC 过渡到真正的 RESTful API 的重中之重。**

这里再讲三个题外话。第一个，**以渐进式思路构造的超媒体格式，是一个被极度低估了的思考模型。**

它的一个具体应用——渐进增强（Progressive Enhancement），是开发前端应用的最佳思考方式，同时也是诸多 JavaScript 框架的核心思路。掌握了它，你就能迅速成为资深前端工程师。

我仍然记得我是如何在领悟了渐进增强之后，在 4 个小时之内，从前端白丁成为前端架构师的。以及，多年前那个初秋的午后，郑晔同学（对，就是在极客时间已经写了三个专栏，正在努力写第四个的郑晔同学）在西安高新区某公交站等出租车时，跟我说起前端学习是如何如何之零碎，不如后端知识系统。当我用了 20 分钟给他讲了渐进增强，然后他几乎是马上就全都会了。

第二，如果通过超媒体关联的资源是代码，那么我们就能指引客户端按照服务端设计的行为，去执行和消费超媒体资源了。

比如，我们现在有一个订单，需要通过支付网关支付。而支付网关可能有很多不同的选择，网银啊、微信啊、支付宝什么的。那么我们可以通过超链接关联支付网关的客户端：

```
2     <title>订单详情</title>
3     <p>....</p>
4     <link href="http://payment.com/12358921" rel="payment"/>
5     <link href="http://payment.geekbang.org/bank.js" rel="payment-client"/>
6 </article>
```

这意味着，客户端不需要知道使用的是哪个支付网关，只需要加载payment-client关联的资源，并通过它完成支付即可（当然我们需要提前约定 JavaScript 的加载策略和入口函数）。于是，服务端进一步控制了客户端的行为，不仅仅是通过超媒体指引了增强的服务，连带如何消费这些服务也提供了指引。

这就是按需代码（Code-On-Demand），通过它，我们能够在将服务器端的计算压力转移到客户端的同时，也不会丧失自己对计算的控制。此外，通过它，也进一步解耦了客户端与服务器之间的耦合。客户端不需要理解服务器端的逻辑，只需要按照指示执行代码即可。可以这么说，按需代码这个技巧，对于需要高灵活度的 API 设计是极其重要的。

第三，这种完全由服务端控制的客户端行为，也被称作 HATEOAS（Hypermedia As The Engine Of Application State），即**超媒体作为应用程序状态的控制引擎**。在 Richardson 成熟的模型中（Richardson Maturity Model），是最高成熟度的 RESTful API。我想经过今天的学习，你应该完全理解其背后的原因了。

好了，我们言归正传，接下来就看看如何使用分布式超媒体来设计 API 中的资源，以实现 HATEOAS。

如何将模型映射为 RESTful API？

通过分布式超媒体设计 API 中涉及的资源

首先我们需要知道什么样的格式属于超媒体。那么**包含超链接的格式都可以看作超媒体**，因此 HTML 是超媒体格式。在前面的例子里，HTML 中的 link 是表达关联资源的主要手段，它的构成要素有两个：

1. 指向关联资源的链接（href）；
2. 与主资源是哪种关联关系（rel）。

这两个是构成超链接的必要因素，缺一不可。同时，只要具备这两个要素，都可以被看作超链接。比如如下 HTML 元素，可以当作超链接：

[复制代码](#)

```
1 <a href="http://payment.com/12358921" rel="payment"/>
2
3
4 <area shape="rect" coords="184,6,253,27" href="http://some-place.com" rel="cen
```

而我们常用的 JSON 和 XML 并没有提供默认的连接格式，所以它们并不是超媒体。于是我们在设计 RESTful API 的时候，要使用 HAL (Hypertext Application Language)，而不是原味的 JSON 或者 XML (Vanilla JSON or Plain XML)。

HAL 目前还是 Internet 标准的一个提案，你也可以自定义自己的 XML 和 JSON 超媒体格式，不过我估计最终结果和 HAL 应该差不多，所以还不如直接用 HAL 来得简单。

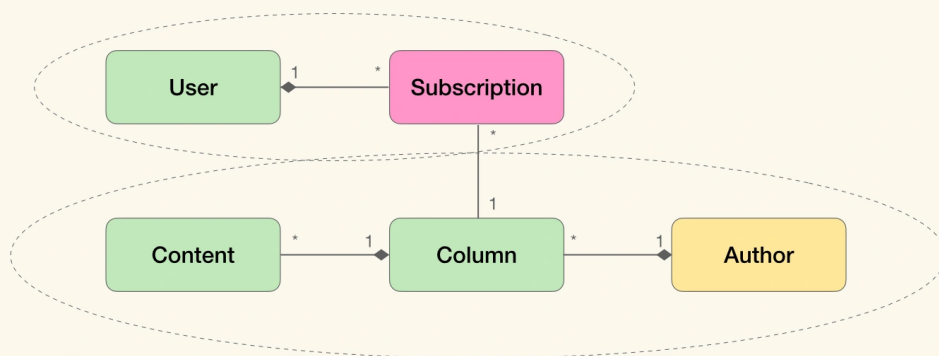
XML 的 HAL 基本上就是把 HTML 的 link 作为标准标签引入了 XML，而 JSON 的 HAL 则定义了 _links 结构来表示链接。如下所示，就是 XML HAL 和 JSON HAL 规范中给出的同一个资源的示例：

[复制代码](#)

```
1 Content-Type: application/hal+xml
2 <resource rel="self" href="/orders/523">
3   <link rel="warehouse" href="/warehouse/56"/>
4   <link rel="invoice" href="/invoices/873"/>
5   <currency>USD</currency>
6   <status>shipped</status>
7   <total>10.20</total>
8 </resource>
9
10
11 Content-Type: application/hal+json
12 {
13   "_links": {
14     "self": { "href": "/orders/523" },
15     "warehouse": { "href": "/warehouse/56" },
16     "invoice": { "href": "/invoices/873" }
17   },
18   "currency": "USD",
19   "status": "shipped",
20   "total": 10.20
21 }
```

解决了格式问题，我们来看看怎么使用超媒体设计 API 中的资源。这里的关键，正如我们刚才所讲，是**如何通过超链接构成渐进式服务消费**。

我们仍以极客时间专栏的模型为例：



首先我们看一下聚合根的超媒体描述。对于一个 User 实例而言，我们可以使用如下的 HAL JSON 来描述它：

[复制代码](#)

```
1 {
2   "_links": {
3     "self": { "href": "/users/19" },
4     "subscriptions": { "href": "/users/19/subscriptions" }
5   },
6   "username": "爱学习的鱼玄机",
7   ...
8 }
```

这段 HAL JSON 中包含两个超链接：self 和 subscriptions。self 表明获取当前资源的 URI，我们会将这个 URI 称作主 URI (Primary URI)。这个 URI 从概念上来讲，和 ID 是等价的，也就是可以用于唯一定位当前资源的标识符。同时，self 也是用以缓存的 URI。

subscriptions 表示了聚合关系 User-Subscriptions，也就是指示了如何寻找被当前 User 聚合的 Subscription。对于聚合根而言，需要为所有的聚合对象提供链接，以指示如何获取这些聚合对象。

你可能会问，self URI 有什么用？如果单独对一个 User 实例而言，可能用处不大。但是在集合资源中，self URI 就是不可或缺的了。比如对于/users，我们使用如下的 HAL JSON 描述：

 复制代码

```
1 {
2   "_links": {
3     "self" : "/users"
4   },
5   "_embedded": {
6     "users": [
7       {
8         "_links": {
9           "self": "/users/18"
10        },
11        "username" : "会聊骚的黄庭坚",
12        ...
13      },
14      {
15        "_links": {
16          "self": "/users/19"
17        },
18        "username" : "爱学习的鱼玄机",
19        ...
20      },
21      ...
22    ]
23  },
24  "total": 43215,
25  ...
26 }
```

可以看到，我们通过_embedded表示了所有的用户。不过这里还有一个需要我们权衡的地方。如果 User 对象有很多属性（比方说包含年龄、家庭住址、个人状态等），那么我们需要在/users的 HAL JSON 中包含所有这些属性吗？

如果包含所有属性，那么这个 HAL JSON 的体积可能会变得非常巨大；而如果不包含所有的属性，那么在客户端需要使用其中某些数据的时候，就会拿不到相应的信息。

这时候就该列表中每一个 User 对象中包含的 self URI 发挥作用了。**我们可以把集合资源和其中的独立资源，看作是渐进式的两种不同服务。**

也就是说，我们可以把在集合资源中包含的数据，看作是更基础的服务，像常用数据这样，可以满足大部分客户端在通常情况下对用户数据的需求。而其中每一个 User 对象中包含的 self URI 所指示的服务，我们可以将其看作是增强服务，也就是全量数据。

于是，如果某个客户端需要不包含在常用数据中的信息，那么它可以通过 self URI 去获取全量数据。**也就是说，我们将常用数据和全量数据设计成渐进式消费的两种不同的服务。并通过分布式超媒体格式，描述了它们之间的关联。**

这样我们就可以让同样一份超媒体描述，来支撑不同的客户端了。与此同时，我们也不需要为不同的客户端提供不同格式的数据信息。当然，你可能会问，这么做，难道不会有性能问题吗？**答案是有，不过没关系，因为有缓存。**

REST 架构大量依赖缓存来缓解性能问题。我们甚至可以说，是否能够有效地利用缓存，会决定 REST 架构的成败。这也是为什么我们强调主 URI 应该是用于缓存的 URI。**也就是在构造资源的时候，我们要将缓存当作必须考虑的特性，详加设计。**

让我通过一个例子，来看看缓存会怎样影响我们的设计决策。这也是非常常见的一个场景——分页。

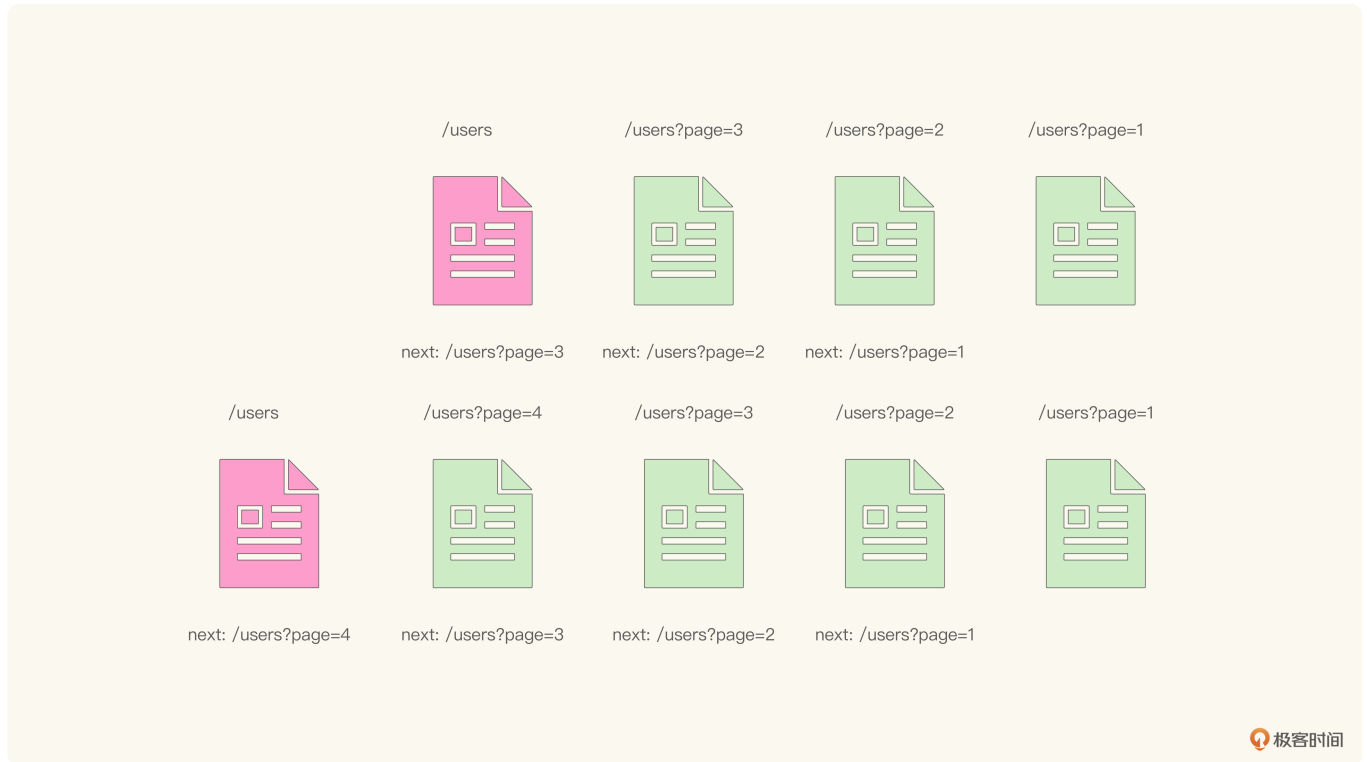
你可能注意到了，在前面的 HAL JSON 中，/users 一共有 43215 个用户，如果将所有的用户都包含进来，就会产生极大的 HAL JSON 文件，因此我们需要分页处理。好在这并不难：

 复制代码

```
1 {  
2   "_links": {  
3     "self" : "/users",  
4     "next" : "/users?page=3"  
5   }  
6   ...  
7 }
```

我们可以通过 next 链接表示后面的一页。那么我们怎么才能通过缓存，使得当有新用户注册的时候，大部分页面的缓存都不失效呢？

一个有效的策略是，永远不缓存/users页面，也就是/users永远表示新近注册的用户。如果它们达到分页上限时（比如 50），就生成新的页面 id，并缓存它，然后继续使用/users等待新注册的用户。如下图所示：



那么前面几页的 HAL JSON 可能就是这样的（如下所示）。而且你会发现缓存下来的页面，我们永远不需要修改其中的内容：

[复制代码](#)

```
1 // 以下为未被缓冲的信息
2 {
3     "_links": {
4         "self" : "/users",
5         "next" : "/users?page=4"
6     }
7     ...
8 }
9
10
11 // 以下为被缓冲的信息
12 {
13     "_links": {
14         "self" : "/users?page=4",
15         "next" : "/users?page=3"
```

```
16     }
17     ...
18 }
19
20
21 {
22     "_links": {
23         "self" : "/users?page=3",
24         "next" : "/users?page=2"
25     }
26     ...
27 }
```

那你可能会有疑问，如果客户端与服务端对分页的要求不同，这时候该怎么办？比如客户端需要每页显示 50 条，而服务器端则需要按每页 20 条缓存？

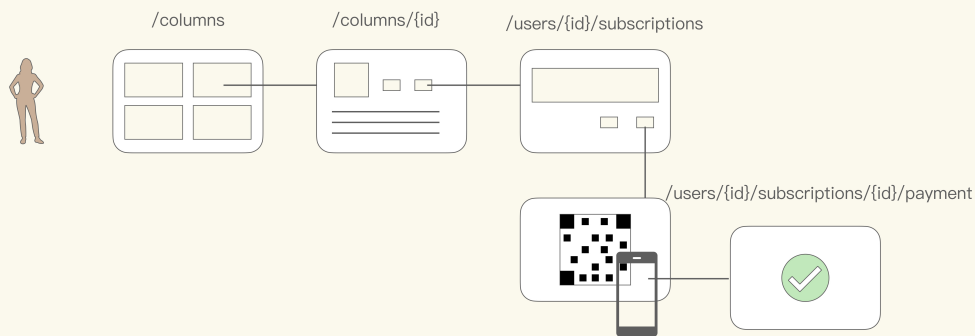
答案是：**服务器永远不需要考虑客户端的需求**。把客户端的额外需求，当作渐进式服务消费的需求，那我们只需要提供对应的链接即可。

也就是说，只要保证页面间的链接，剩下的就交由客户端自行处理。**这就是互联网架构的精髓，将集成与订制推向客户端，从而保持服务端的稳定。**

将 API 映射回业务流程

在我们将 API 中涉及的资源都使用分布式超媒体描述了之后，我们就获得了完整的 API 定义。而且，借由分布式超媒体，我们可以达成 HATEOAS，实现渐进式服务消费，从而充分分离客户端与服务端的耦合与依赖。此外，如果你充分考虑了缓存策略，性能也不会构成任何问题。

那么剩下的最后一步，就是将 API 重新映射回所需要支撑的业务流程或者用户旅程之中，与业务方一起验证这些 API 是否能够满足所有的需求。示意图如下：



如果我们将 API 作为模型的另一种表现形式，那么映射会在流程中与业务方验证，也就和磋商形成统一语言的过程一致了。我想你已经有足够的了解了，就不再赘述了。

小结

这节课我们讲述了什么是分布式超媒体，以及为什么它是互联网架构的默认集成模式。以及我们如何通过使用分布式超媒体形成渐进式服务消费。并以此为核心理念，讨论了如何描述 API 中的资源。此外，我们还重点讨论了集合资源，以及如何通过缓存解决分页等问题，并通过一个分页的例子展示了如何将客户端与服务端充分解耦。

至此，你应该学会了在单体分层架构模式下，如何将模型的能力通过 RESTful API 暴露。那么接下来，我们将进入云时代的业务建模，看看新的架构约束到底将会怎样影响我们建立业务模型。

思考题

你是如何理解云平台的？在你看来，云平台的哪些特性会影响我们的业务模型？

欢迎把你的思考和想法分享在留言区，我会和你交流。那么旧约部分到这里就结束了，我们新约篇章再见！

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

👍 赞 3 💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 10 | 将模型实现为RESTful API (上)

下一篇 说点题外话01 | 好耦合与坏耦合

更多学习推荐

Java 面试必考 300 题
最新汇总

限时免费领取

精选留言 (6)

💬 写留言



大海浮萍 置顶

2021-07-21

看完restful api第二章，还是有些疑问，比如在采购订单场景下，修改订单（修改基本信息）和发布订单（修改状态，通知下游），感觉只能都设计成 PUT /order/{order_id}，那如何区分这两种用例呢，请赐教

作者回复: 修改状态是post 修改信息是put

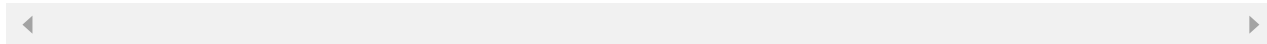
**绝露**

2021-07-21

有没有写过文章，描述那4个小时发生了什么。非常好奇。

展开 ▾

作者回复: 没有

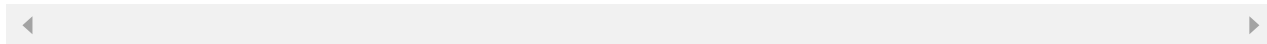
**Oops!**

2021-07-21

我的理解是本节主要讲了RESTFUL API的一个准标准：HAL，前端开发的时候也需要遵循这个标准才能work吧。比如分页的例子，前端按50每页加载，后端20每页的往外吐，那前端就需要理解这个next链接隐含的含义，即next链接返回接下来的数据，但是不一定符合页面分页模式，然后前端需要一直调用next,直到凑足50条，是这样吗？另外，关于缓存，按文中所提模式，如果查询条件比较复杂多变，比如有排序项，过滤等，那这个缓...

展开 ▾

作者回复: 查询一般post 表示不可缓存

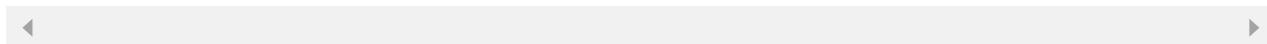
**赵晏龙**

2021-07-19

【服务器永远不需要考虑客户端的需求】这是我在这一章里面学到的最重要的东西，结合你讲的渐进式服务消费，确实能解决一些问题。

展开 ▾

作者回复: www就这么感 在规模优先的系统下 唯一选择

**XXX**

2021-07-19

渐进式服务消费有那么厉害吗。。。

展开 ▾

作者回复: 如人饮水





敬

2021-07-17

渐进式增强和性能如何权衡呢？比如一个page由多个module组成，一个module由多个element组成，如果/pages/{pageid}只返回module的超链接集合，前端要请求多少次啊？

作者回复: 缓存